Programmable Parallel Coprocessor Architectures for Reconfigurable System-on-Chip

John Williams and Neil Bergmann
School of ITEE, The University of Queensland
Brisbane, Australia
{jwilliams;n.bergmann}@itee.uq.edu.au

Abstract

We propose a hybrid rSoC parallel processing architecture consisting of a central 32-bit RISC microprocessor interconnected to an array of 8-bit microcontrollers as coprocessing nodes. The central processor runs an embedded Linux operating system, with the coprocessor nodes mapped into a virtual file system, by which they can be controlled and reprogrammed. The hardware and software architectures are detailed, and several useful application contexts are proposed. Supporting theoretical analysis is also presented.

1. Introduction

Performance predictability is one of the strongest constraints in real-time system design. Unlike transformational computing (e.g. data processing and simulation systems), the feasibility of a real time system is measured by its ability to guarantee that computational tasks will be completed within a certain deadline. The difficulty of assuring real-time performance increases dramatically with the number of interfering tasks [1].

Reconfigurable system-on-chip (rSoC) is a powerful tool for real-time system design. Custom architectures that offload processing from the central microprocessor onto customised hardware or coprocessing units can result in more predictable overall system performance.

In this work we propose a hybrid rSoC multiprocessing architecture consisting of multiple individually programmable microcontrollers interfaced to a central microprocessor system, and consider the subsequent impact on system performance and design flexibility.

The hardware consists of a 32bit softcore CPU, connected in a star topology to multiple dynamically reprogrammable microcontroller cores. We have implemented this architecture using the Xilinx Microblaze processor and its little cousin Picoblaze, as the main and coprocessor respectively. We briefly introduce each of these in Section 2.

The software architecture consists of the uClinux operating system (e.g. [2]) running on the Microblaze, with the programmable state machines mapped directly into the kernel space, and made available to user processes via a virtual file system mapping. We use the term picoware to refer to code running on the coprocessor nodes.

We suggest two specific roles for the architecture. Firstly, by providing an off-chip interface from each coprocessor node, they may be used as intelligent IO processors, offloading main processor load and acting as virtual devices. Low bit-rate communications such as RS232, I2C, SPI, or even low-speed USB can be implemented on the coprocessors. PWM or sigma-delta modulation is another potential use.

The second use-class is for parallelising computational tasks characterised by small data transfers and long computational times. In these cases, the computations can execute in parallel on separate coprocessor nodes. Small block–based cryptographic algorithms are one such computation.

It can be argued that these IO and coprocessing tasks should be implemented more efficiently and compactly in customised hardware, rather than through general purpose programmable coprocessors. We address this point in Section 5, and present a theoretical analysis of the architecture, finally followed by conclusions and directions for future development.

2. Background

Conceptually the architecture is not tied to any particular processor or device, however its implementation as described in this paper has been tested with Xilinx FPGAs and soft-processor cores. In the following we provide supporting background information.

2.1. Microblaze

Microblaze is a compact 32 bit RISC processor, with 32 general purpose registers, and an orthogonal instruction set. It uses a 3 stage instruction pipeline, with delayed branch capability for improved instruction throughput. Microblaze is specifically targeted to logic primitives of Xilinx FPGA devices,

such as hardware multipliers and on chip block RAM (BRAM). Its maximum clock frequency is 125MHz in a modern Xilinx FPGA part [3].

2.1.1. Fast Simplex Links (FSL)

Of particular relevance to the current work is the FSL (Fast Simplex Link) interface. FSL is a unidirectional, point-to-point bus interface [4]. Microblaze has eight each of FSL master and slave ports, named respectively FSL_M/FSL_S 0 through 7. FSL buses themselves are implemented as 32-bit x 16-deep FIFOs, which helps to decouple the timing of the FSL master from the FSL slave. The symmetry of the master/slave interfaces allows FSL-enabled peripherals or processors to be connected in systolic chains or arrays.

In the Microblaze, each FSL port is mapped directly and orthogonally onto the register set. Write operations are performed using

```
put    rS, FSLn
```

where rS is the register containing the source data (r0…r31). Similarly, FSL reads are performed with

```
get    rD, FSLn
```

where rD specifies a destination register.

By default, FSL operations are blocking – writes to a full master FIFO, or reads from an empty slave, will halt the processor until the blocking condition is removed. Non-blocking instructions `nput/nget` are also available. Finally, there is a control bit that may be optionally set and tested, giving rise to the `cput/cget` instructions (and their non-blocking equivalents `ncput` and `ncget`). This allows FSL interactions to be distinguished as data or control operations.

The FSL ports are a powerful interconnect point for coprocessors and hardware accelerators, due to their close integration with the processor and registers, and predictable timing. Non-blocking FSL operations incur a fixed two cycle delay, compared to the shared OPB bus whose transactions may require four cycles or more. Advantages of FIFO-based communications over buses for multiprocessor networks have previously been argued and demonstrated (e.g. [5, 6]).

2.2. Microblaze uClinux

uClinux is a mainstream variant of the Linux operating system with customised memory management code to operate on processors lacking a hardware memory management unit (MMU). Implications of this are no virtual memory or paging, no memory protection, and the absence of fork()'s copy-on-write semantics. Most Linux applications run directly on uClinux, while some that use fork() require modification to the uClinux-compatible vfork() primitive.

In most respects, the Microblaze port of uClinux is very similar to other ports to more conventional processors such as the Motorola Coldfire and ARM devices. A standard Microblaze hardware platform called "mbvanilla" has been developed, with peripherals such as timers, interrupt controllers, memory controllers, GPIOs and an Ethernet MAC building a complete system. Linux device drivers have been wrapped around these cores for interfacing with the kernel and user space applications.

The power and novelty of running Linux on a reconfigurable processor comes from the ability to customise the operating system to support custom hardware architectures. For example, the Virtex-II internal configuration access port (ICAP) has been previously mapped into a Microblaze uClinux system, resulting in dynamically self-reconfiguring Linux systems [7].

2.2.1. Linux and Real-time Systems

Linux is not a real-time operating system. It tends to focus on improving average performance, and does not offer guarantees on metrics such as interrupt latency. Several software based approaches exist to address this shortcoming, most notably RTLinux [8] and the Real Time Application Interface (RTAI) [9]. These systems implement a real-time microkernel executing real-time threads, and simply run the Linux kernel as a low priority process. They also provide communication channels between real-time threads and processes running in Linux.

Our architecture is different, although complementary, to these existing approaches. Rather than pushing real time tasks down into a microkernel layer on the same processor, we push them out to separate coprocessors (as picoware), executing with guaranteed performance.

2.3. Picoblaze

Picoblaze, also known as the K (constant) Coded Programmable State Machine (KCPSM), is a two-operand, 8 bit microcontroller from Xilinx, implemented directly in Xilinx FPGA primitives to minimise logic usage and maximise performance [10]. There several versions available – we use version 2 which has a 1024 x 18bit instruction memory, supporting conditional branches, a return stack for nested subroutine calls, and interrupts. IO is via 256 addressable input and output ports, to which arbitrary external peripherals may be connected. All Picoblaze instructions are completed in 2 clock cycles.

Previous uses of the Picoblaze as a programmable accelerator include implementation as a Field Programmable Port Extender (FPX) module for real-time network packet processing [11], and as a customizable application-specific data and IO processor [12].

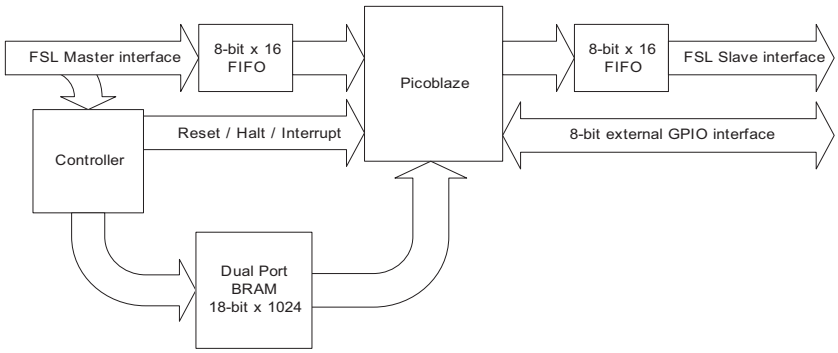

Figure 1. Picoblaze coprocessor node architecture and interfaces

3. Hardware Architecture

In the proposed architecture, Picoblaze coprocessor nodes are connected in a star topology to a central Microblaze, using the FSL links described previously. In the following we describe the coprocessor nodes and interconnect architecture in detail.

3.1. Picoblaze Coprocessor Nodes

The architecture of the coprocessor node is illustrated in

Figure **1**, which serves as a reference for the subsequent discussion.

A controller listens at the input FSL interface of the node. A node control operation is indicated by the presence of the FSL control bit (Sect. 2.1.1). Otherwise, the lower 8-bits of the FSL word are pushed onto the Picoblaze's input FIFO as data for subsequent processing.

Currently supported node control operations are Picoblaze code space write, reset, and interrupt.

Code writes are achieved by the master (Microblaze) sending a control command to set a write-address register, which is then automatically incremented on each subsequent opcode write. This seek/sequential programming model efficiently supports both bulk code writes (reprogramming entire code memory), and small changes such as updating individual data items encoded in the Picoblaze opcodes. It is advisable, although not mandatory, to hold the node in reset during code updates.

Two 8-bit x 16 deep FIFOs are interfaced onto the Picoblaze's IO space, one each for read and write. The FIFOs can be read and written by Picoblaze in either blocking or non-blocking modes, to mirror the FSL semantics (Sect. 2.1.1). A "Halt" signal was added to the standard Picoblaze core, that forces an extended T-state if a blocking write is attempted on a full output FIFO, or a blocking read on an empty input FIFO. Halt is asserted by the node controller if the input FIFO is empty and the Picoblaze attempts a blocking read operation (and respectively blocking write on a full output FIFO). The Picoblaze may test the FIFO status, to determine whether an operation would block.

The introduction of the FIFOs and optional blocking/non-blocking operations to the process nodes supports elegant solutions to host-coprocessor synchronization issues. Examples are presented later in the paper.

A final feature of each coprocessor node is an 8-bit direction-programmable GPIO port. The GPIO would typically be mapped to external pins of the FPGA, however it may also be connected to other logic in the design if required. The GPIO inputs, outputs, and direction control registers are mapped into the Picoblaze IO space.

Logic usage statistics for a single coprocessor node, comprising of a Picoblaze, controller, FIFOs and GPIO are shown in Table 1. For multi-node systems, the limiting factor is clearly the total number of FSL ports available, and also on-chip Block RAM (BRAM) usage. Each Picoblaze node requires one full BRAM for its instruction memory.

In these experiments, a complete uClinux-capable Microblaze system, including Ethernet MAC, instruction/data caches and 8 Picoblaze coprocessor nodes, fit comfortably into a Xilinx Virtex2-1000 part (approx 75% logic utilisation). The development board used is a standard Insight/Memec V2MB1000 Microblaze development board.

3.2. Microblaze to Coprocessor Hardware Integration

Each coprocessor node presents one each of an FSL master and slave interface. These interfaces are connected directly to a Microblaze master and slave FSL port. Microblaze currently supports 8 FSL channels, and thus up to 8 coprocessors may be connected, as illustrated in Figure 2.

Microblaze communicates with each node either

Table 1. Logic usage – single coprocessor node

```
Selected Device : 2v1000fg456-4

Number of Slices:           151 out of  5120 2%
Number of Slice Flip Flops: 114 out of 10240 1%
Number of 4 input LUTs:     221 out of 10240 2%
Number of BRAMs:              1 out of    40 2%
```

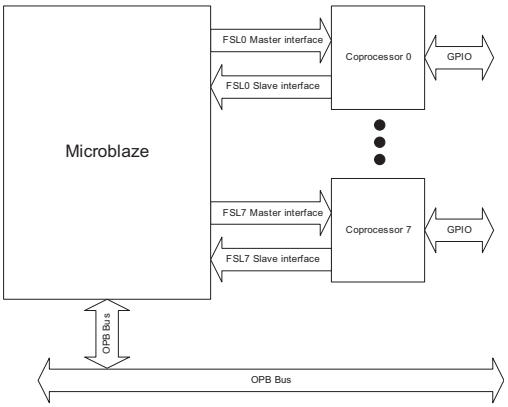

Figure 2. Microblaze and coprocessor node interconnection

through control operations as described previously, or as regular data operations. Data writes to a master port map connect to the node input FIFO, while reads access the node output FIFO.

Master/slave symmetry between read and write operations is preserved – just as a node may optionally block, so too can Microblaze, with the blocking get/put operations described in Sect. 2.1.1.

In practice, it is highly inadvisable for Microblaze to perform blocking operations – a stalled coprocessor could deadlock the entire system. This is particularly the case when the Microblaze is running a multitasking operating system like uClinux. The uClinux integration of the processor architecture forbids instruction-level blocking. Instead, user processes may elect to block, but at the device driver level the FIFO status is polled periodically, rather than utilizing hardware blocking get/put operations.

4. Operating System Integration

The hardware architecture just described can be used directly by low level Microblaze software. The procedures to program, control, and communicate with each node are simply combinations of FSL `put` and `get` instructions. Table 2 outlines the primitives of the library developed to support low level coprocessor node communications and control.

This API could also be used directly by user-space uClinux programs. However, we chose to map the architecture into the uClinux kernel, representing each Picoblaze as a system resource that may be acquired, read, and written. We expand on this mapping below, before offering justification for this approach over a potentially lighter-weight direct programming model.

4.1. The Linux Virtual File System

Like most modern operating systems, Linux uses a virtual file system (VFS) abstraction model, beneath which exist physical file system implementations. For example, the Network File System (NFS) presents the same interface as does a light-weight ROMFS commonly used in embedded Linux systems.

Table 2. Low-level coprocessor management API

```
int pico_init(struct pico_data_t *pico, int
num);

int pico_get_reset(struct pico_data_t *pico);
int pico_put_reset(struct pico_data_t *pico,
        int reset_val);

int pico_interrupt(struct pico_data_t *pico);

int pico_code_read(struct pico_data_t *pico,
        off_t offset,  unsigned *buffer,
        size_t count);
int pico_code_write(struct pico_data_t *pico,
        off_t offset, unsigned *buffer,
        size_t count);

int pico_data_read(struct pico_data_t *pico,
        unsigned *data, unsigned *status);
int pico_data_write(struct pico_data_t *pico,
        unsigned data, unsigned *status);

int fsl_blocked(unsigned status);
```

This has some powerful benefits – the underlying file system is completely transparent at the application level. In the dynamically self-reconfiguring Linux work mentioned previously [7], networked dynamic self-reconfiguration was achieved trivially by mounting a remote host system via NFS, and processing partial bitstreams as though they were regular local files.

VFS also gives rise to truly virtual file systems which have no underlying physical manifestation but instead are constructed dynamically in response to process requests such as reads, writes, directory listings and so on. The best known use of this capability is the Linux proc file system (procfs). Normally mounted into the /proc directory, procfs provides a window into the internal operations of the kernel, allowing processes to inspect and in some cases modify the operation of the kernel.

4.2. Coprocessor array mapping into procfs

In accordance with the interpretation of the coprocessor nodes as configurable system resources, we choose to map these nodes directly into the procfs. This is achieved by writing a simple file-like interface wrapper around the low-level API shown in Table 2. The procfs interface is implemented as a loadable kernel module, which creates the virtual directory and file structure upon initialization.

The virtual directory structure is */proc/picoblaze/pico0* through to */pico7*. Within each Picoblaze virtual directory exist three virtual files, and one virtual device

- **reset** – writing a non-zero value to this file places the Picoblaze in reset. Writing a zero releases reset. Reading this file returns the current reset state.

- **interrupt** – writing a non-zero value triggers the Picoblaze's interrupt line. A read operation has no meaning, and always returns zero.
- **code** – a binary virtual file that maps the program memory of the Picoblaze node. This file may be read or written, and the seek operation is also defined.
- **data** – a Linux character device node. Write operations place data into the Picoblaze's input FIFO, and reads extract data from the output FIFO. This device is discussed in greater detail below.

Before expanding on the details of this structure, it is illuminating to point out some implications of this mapping, using some simple examples.

4.2.1. Programming a node

A coprocessor node is programmed simply by writing binary opcode data into its */code* virtual file:

```
$ cat pulse_pwm.hex >
     /proc/picoblaze/pico0/code
```

4.2.2. Copying a coprocessor process

A coprocessor "thread" executing on one node, may be duplicated onto another node:

```
$ cp /proc/picoblaze/pico0/code
     /proc/picoblaze/pico1/code
```

This example is not true process duplication, since no state information is transferred. However, with appropriately coded Picoblaze software, this could be achieved:

```
$ echo 1 >
     /proc/picoblaze/pico0/interrupt
$ cat /proc/picoblaze/pico0/data >
     /proc/picoblaze/pico1/data
```

In this example, the picoware is coded such that when an interrupt is received, it outputs its current state information. Thus, by copying the code space and then interrupting the first "thread", its state dump is redirected into the new thread, which can then continues where the first left off.

4.2.3. Cascading two nodes into a sequence

Two coprocessor nodes may be cascaded together:

```
$ cat /proc/Picoblaze/pico0/data >
       /proc/Picoblaze/pico3/data
```

This sort of approach would be useful if a particular algorithm was sequentially partitioned between two nodes. The intermediate outputs from one stage are passed on to the next for completion.

While these examples are presented in simple shell script, the operations can just as easily be performed from C programs, accessing the virtual files just like regular files with open(), read() and write() library calls.

4.3. The FIFO Device Driver

The /proc/picoblaze/pico*N*/data device is a regular Linux character device node, that implements kernel level IO buffering. This is in addition to the hardware buffering provided by the 16-deep input and output FIFOs on each processor node. In its present incarnation the Picoblaze node is not able to generate Microblaze interrupts, thus kernel tasklets are used to poll the physical FIFO status and transfer data between kernel buffers and the hardware FIFOs.

The reasoning behind the additional kernel layer of buffering is the same as for more conventional system devices such as disk units – buffered IO can reduce the overhead of excessive process-kernel interaction by allowing larger chunks of data to be transferred between processes and the kernel at a single time. Maintaining separate buffers for each node also eases the problem of scheduling reads and writes across multiple nodes.

If the coprocessor nodes were to be accessed directly (the light-weight approach mentioned at the beginning of this section), rather than the kernel buffering model, significant custom software would be required in order to efficiently schedule IO operations to the multiple nodes.

The naïve approach – polling each device in a round robin fashion, each time reading as much data as was available, and writing as much data as could fit –would be very inefficient, particularly if the data transmission rates varied across the nodes.

We can easily trade between software and hardware buffering at compile/synthesis time. The hardware FIFOs are implemented in the Virtex-II SRL16 primitive, and can easily be cascaded together to provide arbitrary depths (up to the capacity of the chip). Larger hardware buffers would provide a measurable, although asymptotically limited performance improvement, by increasing the amount of data that can be transferred from the kernel to the hardware on each transfer cycle, and thus reducing task switching (or, potentially, interrupt handling) overhead.

5. Analysis and discussion

We present an analysis of the architecture, first by outlining some of its useful characteristics, from which we propose two classes of application that can most benefit.

5.1. Useful characteristics

The following considers some of the characteristics of the architecture and its implementation that are useful in the context of real-time embedded systems.

5.1.1. Improved predictability

As mentioned previously, one of the greatest challenges facing real-time system designers is the ability to guarantee computational deadlines. For conventional microprocessor systems, this commonly requires significant over-engineering of the central processor, to cover a rare conjunction of events (the worst case scenario).

The implementation described in this paper uses Picoblaze coprocessors clocked at 66MHz, executing a consistent 33 million instructions per second, irrespective of what the central processor or other coprocessor nodes are doing.

Migrating real-time tasks onto these coprocessor nodes has two positive side-effects:

- Task execution predictability is improved
- Central CPU load is reduced

Communication between central and coprocessor nodes must still be considered, and ideally should be minimised to take maximum advantage of the architecture.

5.1.2. Logic reuse

Once deployed, an embedded system may require only very infrequent use of some external communication peripheral, such as a serial interface, for in-the-field testing or maintenance. In cost and power sensitive applications it is wasteful to dedicate logic area to a peripheral device that may only be used with a duty cycle of perhaps 1% or potentially much less.

FPGA power consumption is strongly influenced by static leakage currents, which are present even if a particular logic module is not actively switching. The only way to achieve significant power reduction is to fit the application into a smaller device.

The coprocessor node(s) of rarely used IO functions implemented as picoware can be executing other useful tasks when not required for the infrequent IO operations. This is a form of run-time logic reuse – the logic cost of the coprocessor is amortised over the entire application execution time.

Dynamic and/or partial reconfiguration is commonly argued as a means of achieving run-time logic reuse. While this is true in principle, in practice the technique is inadequately supported by design tools and FPGA devices. Significant extra design effort is introduced to meet the floorplanning and inter-module signal routing requirements of partial-reconfiguration support. The network of programmable coprocessors may not have the same performance or flexibility as a customised hardware, however it is dramatically simpler to use.

Depending upon the degree of dynamic reconfiguration, the functional switching time of the reprogrammable architecture may also be faster. It is certainly of finer granularity, with individual opcodes in a coprocessor program able to be modified.

5.2. Roles for the architecture

Clearly, an 8-bit Picoblaze will not outperform a dedicated 32-bit Microblaze at the same clock frequency. Meaningful roles for the coprocessor network architecture will leverage the available parallelism, and the subsequent predictability improvements resulting from decreased load on the central processor.

We propose two roles for the architecture. The first is for intelligent virtual IO devices, and the second is for parallel computational coprocessors. We discuss each below, and consider specific constraints that influence the feasibility of such an approach.

5.2.1. Intelligent peripherals

The consistent instruction throughput of the Picoblaze and its relatively high performance (in excess of 50 8-bit MIPS) makes it suitable for a range of IO tasks. The fixed instruction throughput permits software timing loops that would be either impossible or wasteful on the central processor.

Medium bitrate communications – Serial IO standards like RS2323, SPI, I^2C, and even low speed USB can be implemented in a coprocessor node. In some instances, for example SPI, the Picoblaze node has a smaller logic footprint than an equivalent logic core [13], resulting in an overall net logic reduction.

Low bandwidth IO controllers –an 8-bit PWM output controller was implemented on a Picoblaze node. This virtual device has an output bandwidth of 16KHz at 66MHz (2056 Picoblaze instructions per PWM cycle) – adequate for servo motor control or low quality audio.

Intelligent IO controllers – Many legacy embedded systems and devices, as well as modern units like GPS modules use low bit-rate RS232 for serial packet-based communications. Some miniature integrated sensor devices like current and temperature sensors use PWM and/or I^2C to initiate sensing and communicate sensor values. All of these interfacing functions can be offloaded to a coprocessor node.

5.2.2. Coprocessing

In the role as a parallel coprocessor for, the most restrictive constraint is communications bandwidth. If Microblaze takes longer to communicate data to the coprocessor than to perform the actual computations, the no performance improvement is gained. However it may still make sense to offload the processing, if predictability is improved. A candidate for this role is low rate, high complexity encryption. Picoblaze has been demonstrated performing AES encryption [14], and although at low speeds, that encryption would come at effectively zero cost to the central processor.

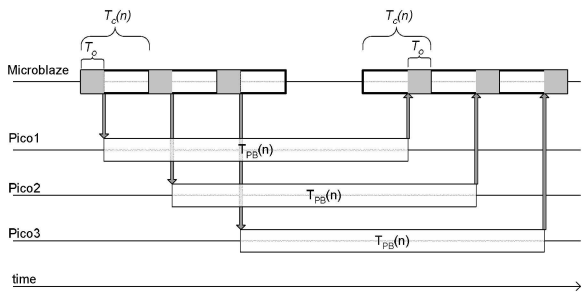

Figure 3. Simplified parallel processing sequence for theoretical modeling

5.3. Theoretical analysis

In this section we present a simple mathematical model and analysis of the architecture. Certain reasonable assumptions are necessary – the intention is establish identities and inequalities that describe the range of useful applications.

We assume that computational tasks are discretised such that each task involves communication of, and operations on n individual data items, and that there are N such tasks to be computed. Let

$T_c(n) = T_o + nT_d$

be the time of communicating n data items between the central processor and a single node, T_o is the fixed overhead (e.g. system call entry), and T_d is the time to transmit one data item.

Next, let $T_{PB}(n)$ be the time for a single Picoblaze node to perform a computation on those n data items, and let $T_{MB}(n)$ be the time for a Microblaze to perform the same computation, also on n data items.

For simplicity we assume that Microblaze and Picoblaze implementations use algorithms with the same underlying complexity $\Omega(f(n))$. Then, let

$$K = T_{PB}(n)/T_{MB}(n) \quad (1)$$

be the sequential speedup of the Microblaze over Picoblaze for the specific algorithm (e.g. due to word length differences and other processor architecture benefits).

Finally, let P be the number of coprocessors available, and we assume that this is equal to N, the number of tasks. That is, we do not consider the problem of scheduling N tasks onto P processors if $N \neq P$.

Figure 3 illustrates the processing sequence of a central Microblaze communicating with N=3 nodes. The Microblaze spends time $T_c(n)$ communicating data to each node, each of which takes $T_{PB}(n)$ to complete. Finally, the communication is repeated to return data to the Microblaze. In this model, the nodes are being used as transformational coprocessors, producing as much data as they are fed. This is one of the worst-case scenarios for the architecture, since it doubles the amount of inter-node communication.

Because of the FIFO based communication, we assume that each Picoblaze task can start as soon as a single item has been sent, that is after T_o+T_d delay. Similarly, the central processor finishes receiving its data with a delay T_o after the corresponding Picoblaze completes processing.

Finally, we assume that no Picoblaze task completes before the Microblaze has completed writing data to all tasks, or approximately $T_{PB}(n) > N.T_c(n)$. This only holds for relatively cheap communications and small numbers of nodes P. In the current system we are limited to 8 nodes, so consider this to be a reasonable approximation.

For comparison, the sequential time required were Microblaze to perform all of the computation is generously approximated as the product of the number of tasks and the time to perform one task (i.e. assuming zero communication overheads):

$$T_S = N,T_{MB}(n) \quad (2)$$

From Figure 3, execution time for the parallel system is

$$T_P = 2(T_o+T_d)+(N-1)(T_o+nT_d)+T_{PB}(n) \quad (3)$$

For ease of analysis we approximate (3) as

$$T_P \approx N(T_0+nT_d) + T_{PB}(n) \quad (4)$$

From (4) we identify three different classes of tasks

- **Communication-bound:**
  $$N(T_0+nT_d) >> T_{PB}(n) \quad (5)$$
- **Compute-bound:**
  $$N(T_0+nT_d) << T_{PB}(n) \quad (6)$$
- **Mixed** – neither dominates.

We consider two metrics – the first is the speedup S, the ratio between the parallel and sequential processing times. It represents the overall speedup achieved by a parallel implementation:

$$S = N.T_{MB}(n)/(N(T_o+nT_d)+T_{PB}(n)) \quad (7)$$

For compute-bound systems (6), this simplifies to

$$S \approx N.T_{MB}(n)/T_{PB}(n) = N/K \quad (8)$$

Thus, for compute-bound systems, the proposed architecture will result in an overall speedup (S>1) if the number of coprocessor nodes N exceeds the sequential speedup of a Microblaze implementation over a Picoblaze implementation (K). Since N is bounded, the scalability of this speedup is limited.

For communication bound systems the speedup approximates as

$$S \approx T_{PB}(n)/K(T_o+nT_d) \quad (9)$$

However, communication-bound implies Eqn. (5), and thus we have

$$S << 1 \quad (10)$$

This result is unsurprising – a communication bound system spends too much time moving data around, and not enough time actually performing work on that data.

The theoretical analysis shows that migrating heavily compute-bound functions onto coprocessor nodes results in significant offloading of the central processor, and confirms that communication-bound processes are to be avoided.

6. Conclusions and Future Development

We have presented an architecture for parallel processing of computational and IO tasks on programmable microcontrollers linked to a central microprocessor in a star topology. We have further shown how this coprocessor array may be naturally and efficiently integrated into the Linux operating system executing on the central processor, and that such a mapping provides uniform and simple access to the coprocessor array.

Performance predictability – the impact on central CPU load is minimised by mapping real-time tasks onto coprocessor nodes.

Flexibility – a node with access to an external IO port may be used as a regular computational node when not required for IO duties. Coprocessing tasks not dependent on external node connectivity may be executed on any available node.

Simplified logic re-use – the simplified programming model offers some logic re-use advantages of partial reconfiguration without the excessive complexity.

The proposed architecture can just as easily be applied to systems with more powerful coprocessor nodes. Indeed, with increased logic usage the architecture could be inverted, placing a simple microcontroller at the heart of an array of powerful microprocessors.

Future developments in the work will include experiments with different types of coprocessors, including custom nodes with greater support for numeric processing such as DSP operations, as well as investigations to see how the coprocessors themselves may be used to accelerate specific operating system functionality. More broadly, we are investigating a wide array of single-chip multiprocessing architectures, and their mappings into the Linux operating system paradigm.

7. Acknowledgements

The support of the Australian Research Council is gratefully acknowledged. The authors would also like to thank Goran Bilski for advice on the implementation of the ideas presented in this paper.

8. References

[1] N. W. Bergmann, G. Brebner, and J. P. Gray, "Reconfigurable Computing and Reactive Systems," in Proc. Australasian Workshop on Parallel and Real-Time Systems (PART '00), Newcastle, Australia, 2000.

[2] A. Rubini and J. Corbet, Linux Device Drivers, 2nd ed: O'Reilly and Associates, 2001.

[3] Xilinx, "Microblaze Processor Reference Guide," Xilinx, Inc, 2003, pp. 136.

[4] Xilinx, "Fast Simplex Link Channel," Xilinx, Inc., San Jose, CA, Product Specification DS449, 22nd Mar. 2004.

[5] G. Kahn, "The semantics of a simple language for parallel programming," in Proc. IPIF '74, pp. 471-475, Amsterdam, 1974.

[6] C. Ross and W. Bohm, "Using FIFOs in Hardware-Software Co-Design for FPGA Based Embedded Systems," in Proc. IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 2004.

[7] J. A. Williams and N. W. Bergmann, "Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip," in Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04), Las Vegas, Nevada, 2004.

[8] M. Barabanov, "A Linux-based Real-Time Operating System." Sorocco, New Mexico: New Mexico Institute of Mining and Technology, 1997, pp. 43.

[9] P. Mantegazza, "Dissecting RTAI," online at http://www.aero.polimi.it/~rtai/documentation/articles/paolo-dissecting.html, accessed 7th June 2004.

[10] K. Chapman, "PicoBlaze 8-Bit Microcontroller for Virtex-II Series Devices," Xilinx, Inc., San Jose, Application Note XAPP627, 4th Feb. 2003.

[11] H. Fu and J. W. Lockwood, "The FPX KCPSM Module: An Embedded, Reconfigurable Active Processing Module for the Field Programmable Port Extender (FPX)," Washington University, Department of Computer Science, Technical Report WUCS-01-14, July 2001.

[12] U. Bidarte, A. Astarloa, A. Zuloaga, J. Jimenez, and I. M. d. Alegria, "Core-Based Reusable Architecture for Slave Circuits with Extensive Data Exchange Requirements," in Proc. Field Programmable Logic and Applications (FPL '03), Lisbon, Portugal, 2003.

[13] Xilinx, "OPB Serial Peripheral Interface (SPI)," Xilinx, Inc., San Jose, CA, Data Sheet DS464, 29th July 2004.

[14] Mediatronix, "Picoblaze code for Rijndael (AES-128) block cipher," online at http://www.mediatronix.com/examples/Rijndael-2.htm, accessed 7th June 2004.